VSCODE EXTENSION FOR LC-3 PROGRAMMING

BY

QI LI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science in Computer Engineering
in the College of Engineering of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Adviser:

Associate Professor Steven Lumetta

ABSTRACT

As an aid to the teaching and learning processes in elementary programming classes, we developed a VSCode extension to provide static-analysis-based feedback for LC-3 assembly code. The extension implements per-instruction analysis, control flow analysis, dataflow analysis, and subroutine analysis. Feedback messages about potential issues in the code are conveyed via VSCode's squiggles and pop-up windows. The ability to convey meaningful feedback messages to students while they write their code increases the expected functionality grade in machine problems. For the first machine problem in ECE 220, within all commits that assemble, the code samples that are warning-free have 10 points higher average functionality score than do commits that generate warnings.

The extension code can also be used as a standalone program to process source code in bulk and to provide detailed information on issues. This program can be used by teaching staff to obtain information on all students' code, potentially reducing the human time involved in the evaluation of coding style. Using the program, we also found evidence that students using the VSCode extension tend to have fewer issues than those who did not use the extension, suggesting that students do try to address issues brought to their attention. Comparing across solutions produced by different classes reveals that the tool is also able to identify interesting stylistic variations likely induced by slight changes in the assignments, indicating a need for more careful instruction on certain topics—in the case found, the implementation of loop constructs. Students doing an assignment with slightly greater difficulty in designing loop structures were 34% more likely to hand-unroll loops. At the time of deposit, the extension has been downloaded by 206 users, not including students in Fall 2020.

Subject Keywords: Static Analysis; Assembly Language; Software Engineering; Student Feedback

ACKNOWLEDGMENTS

I would like to express my gratitude to my adviser, Prof. Steven Lumetta, for all the insightful comments and remarks on the thesis. Your expertise inspired me in formulating the methodology and pushed me to sharpen my mind and brought my work to a higher level.

I would also like to thank my classmates, Zikai Liu, Tingkai Liu, and Wenqing Luo for their collaboration and discussion, as well as for their work on the other parts of the automatic feedback system.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Learning to program is difficult, especially when novice programmers do not receive immediate feedback on their work. While instructors and teaching assistants (TAs) try their best to provide helpful feedback to students, the feedback is usually not given in a timely manner. Also, in introductory programming classes, such as ECE 120 and 220, the class sizes are usually big, adding difficulty to give individual attention to each student's programs. For ECE 220 offered in Fall 2020 at the ZJU-UIUC joint institute (ZJUI), the staff developed an automatic feedback system to address the need for rapid feedback to each student. In the class, students first program in assembly language for the LC-3 instruction set architecture (ISA), which was invented for educational purposes in the textbook by Patt and Patel [1].

As an introductory course, roughly a third of the grade for each assignment is based on a student's coding style. It is common for introductory courses to weigh heavily on coding style. In ECE 120 for which the author served as TA in Spring 2018, roughly 45% of the points in each of the two programming labs are based on coding style. The reason for stressing the importance of style is that we are trying to help students to develop a good coding style early. Advanced courses offered for junior and senior students instead usually put all weight on functionality. Students who have not developed a good style by that point in their education typically cannot complete the required functionality.

While calculating the functionality component of a grade takes seconds using computer scripts, grading style takes several minutes for each student. As a teaching assistant in ECE 220, the author participated in the grading of student programming assignments. During manual grading of the assignments, we found that lots of students lose points because of stylistic issues, such as insufficient comments and an inability to formulate loops.

Using a program to automatically detect stylistic issues can potentially save time in grading, leaving more time for staff to help students in other ways. Also, it is more painful for students to fix their stylistic issues after they have completed a whole assignment. Giving warnings while students write their code encourages students to fix any issues immediately, rather than realize the problem after they submit their code.

To enhance the effectiveness of encouraging students to form a good coding style and give students tailored feedback on their code, we developed a VSCode [2] extension that enables both functional and stylistic checking of LC-3 programs and provides immediate feedback to students as they edit their code. We also integrated basic functionalities common to modern extensions for language support, such as Python and C, such as syntax highlighting, tab completion, and one-click jump to label definitions. The extension is available publicly to all VSCode users at the VSCode marketplace [3] and has been downloaded by 206 users (about twice the number of students in the Fall 2020 class, to whom the extension was distributed privately using installation packages).

The VSCode extension is a part of a bigger system built collectively by the staff of ECE 220 in Fall 2020. The system aims to automatically generate feedback to each student based on the code they write and the assignment specifics. The core of the feedback system [4] is built on the low-level infrastructure provided by KLEE [5]. We implemented our own modules for LC-3 so that we can use them for LC-3 program checking. Students start to use the system when they edit their code in VSCode using the extension described in this thesis. After they finish the code, they submit the code through Git [6] and receive the analysis and feedback results in another branch of their repository [4], [7]. We also developed a reverse debugging tool [8] and a web interface [7] to improve the debugging experience of students.

We deployed the system to give feedback to over 100 students taking ECE 220 in Fall 2020. Each student implemented three LC-3 assignments over four weeks. Each assignment includes the code from the previous assignment directly and builds on it. In the first assignment, which we call MP1, students were asked to write two subroutines to print formatted strings to the display. In the second assignment, MP2, students read a list of events and put them in a schedule, then print the schedule in a weekly

format to the display. Each event consists of a name, an hour, and a bit vector of the days in a week on which the event occurred. In the third assignment, MP3, students implemented a depth-first search (DFS) algorithm to fit more events with optional hours (encoded in a bit vector) into an existing schedule. Students wrote a median of 693 lines of LC-3 code for MP3 assignments (the codes of the first two assignments are counted).

Since students received feedback each time they pushed their code to the GitHub server, they tended to commit more frequently as they wrote the code, providing us with 1079 code samples for MP1, 1474 samples for MP2, and 995 for MP3.

The thesis is organized as follows. Chapter 2 shows how the VSCode extension performs various functions and provides details of each function. Chapter 3 describes several analyses that we implemented and explains several interesting aspects of the analysis designs. Effectiveness and analytical results of the extension are evaluated in Ch. 4. Chapter 5 concludes the thesis.

CHAPTER 2

FUNCTIONALITY

The VSCode editor offers a set of tutorials for developing extensions on the extension guide website [9]. We utilized the code skeleton there as a starting point.

The extension contains three parts: a syntax highlighting engine, a client, and a language server. The syntax highlighting engine tokenizes the text and highlights tokens with different colors according to the color scheme chosen by the user. The client and server serve as the front-end and back-end, respectively. The client and server communicate through the Language Server Protocol (LSP) [10]. An example is shown in Fig. 2.1.

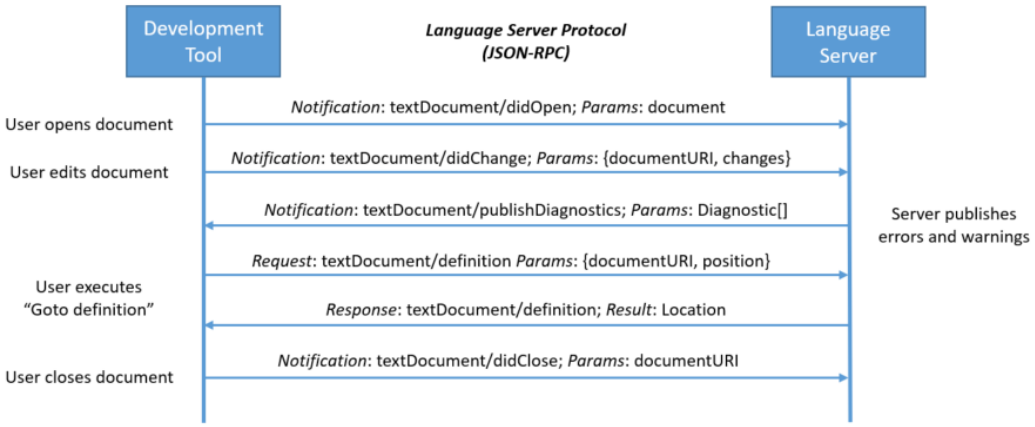

Figure 2.1: An example of communication between a client and a language server during a routine editing session. [10]

The client sends requests to the server when specific events occur. The language server executes tasks corresponding to each request and returns responses to the client, which are then displayed to the user via the client's graphical user interface (GUI).

For example, whenever the code changes, the client sends a request with the change to the LC-3 language server. The server performs various checks on the code, and sends the feedback to the client. The client then shows the three kinds of feedback: errors, warnings, and information. Errors imply that the code cannot assemble. Warnings indicate potential bugs or poor style. Information shows the results of analyses on the code, such as which registers in a subroutine are callee-saved (have their values preserved by the subroutine). Feedback messages about potential issues in the code are conveyed via VSCode's squiggles and pop-up windows.

In the rest of the chapter, we introduce the functionality that we implemented in the extension.

2.1 Syntax Highlighting

We implemented syntax highlighting for LC-3 by tokenizing the code into opcodes, registers, numbers, and strings and assigning different colors to them. We specified the rules for tokenization using regular expressions. The actual color varies based on the color scheme chosen by the user for VSCode.

2.2 One-click Jump to Label Definitions

One-click jump to label definitions is a helpful function when viewing long code with labels far from each other. The extension keeps track of all label definitions with the line numbers in a list. When the user clicks on a label while holding Ctrl, the name of the label is found in the list. If the label is present, the cursor moves to the label definition.

2.3 Auto-completion of Keywords and Labels

Similar to Sec. 2.2, the extension keeps track of all labels in the code. In this case, both definitions and references are recorded to cover the case where the user types the reference of the label first and then defines the label below. Also, all opcodes and predefined pseudo-ops are included in the list of completion candidates so that they can also be auto-completed.

There are possible improvements in auto-completion, though. Currently, the approach that we take does not consider the context of the word being auto-extended. For example, if the user types "LD R," the operand being typed must be a register. However, labels starting with R are also shown in the list of autocompletion candidates. This aspect could be improved.

2.4 Static Checking on the Code

The most powerful aspect of the VSCode extension is the static checking on the code. For code examples and specific feedback messages for each type of issue, please refer to Appendix A.

Illegal instructions: Instructions that are incomplete or illegal generate errors. We parse student code in the extension, find any instructions with an incorrect number of operands or the wrong type of operands. Each illegal instruction is marked as an error.

Immediate values: Ranges of immediate operand values are checked. If they exceed the limit of the specific instruction, an error is generated. For AND and ADD instructions, an additional check on whether encoding issues occur is performed. A warning is generated when a positive decimal immediate number is used, but the encoding makes the value negative. This rule is explained more thoroughly in Sec. 3.1.

PC Offsets: Ranges of PC-relative offsets are checked. A warning is generated when the value is too large. If students write hard-coded offsets rather than using labels, a warning is generated. Writing hard-coded offsets is error-prone and reflects poor style, so we encourage students to use labels.

Control flow: If control reaches a HALT instruction while executing a subroutine, or reaches a RET instruction while executing the main code, a warning is generated. Incorrect use of HALT or RET instructions indicates that the programmer does not understand the difference between the main program and a subroutine.

Start address and end of file: Code before the .ORIG directive is shown as errors. Code after the .END directive is shown as warnings.

Data execution: Code that potentially executes data as instructions generates warnings.

Code overlap: Shared code between the main code and/or subroutines generates warnings. Although sometimes sharing code between subroutines can be useful in reducing code size, it makes debugging harder and undermines the extensibility of the code.

Label checking: Label names that are not usable, such as X10, R3, and 12LABEL, generate warnings.

Duplicate labels cannot appear in the same file. An error is generated for each label that appears earlier in a file.

Having multiple labels at the same memory location is not a fatal issue, but may illustrate structural defects in the code. All labels but the last one are marked as unnecessary (grayed out).

Condition codes: If the conditions used for a branch instruction are always true or always false, a warning is generated. Although writing code with redundant condition codes does not affect the correctness of the code, it can confuse another programmer reading the code, and can lead to problems when trying to analyze the control flow. This complexity undermines the effectiveness of other checks, such as those performed for reachability and dead code.

Unreachable instructions: Unreachable instructions are instructions that can never be executed unless one manually sets the Program Counter (PC) to the corresponding address. When control flow causes instructions to be unreachable, those instructions are marked as unnecessary.

Dead code: Instructions that produce results that are overwritten without being used are called dead code. The tool performs recursive data flow analysis to identify such instructions. Dead code is then marked as unnecessary. In some cases, students may not have finished writing the code that uses results from dead code, or may have written bugs that break the intended dataflow. Students are encouraged to either fix such problems or remove the dead code to improve their code.

Subroutine analysis: The extension reports the callee-saved registers in a subroutine if they are preserved using consecutive STs right after the subroutine label and restored using consecutive LDs right before an RET instruction. Any mismatch of saving and restoring registers generates a warning.

Uncalled subroutines: Uncalled subroutines are marked as unnecessary. To account for the false-positive situation in which a student writes a subroutine before calling that subroutine, or in which the instructor's code calls the student's subroutine, the extension provides a way to prevent the warning by adding a comment "; @SUBROUTINE" in the line right above the subroutine label. This action can also be performed through a VSCode "quick fix"—a button that appears when a user hovers the mouse cursor over the subroutine label.

Unrolled loops: Consecutive repeated code blocks are recognized as an unrolled loop, which means that a loop may be used in place of the repeated code. A warning is generated for each unrolled loop. It is a good practice to write loops or subroutines to perform repetitive work rather than copying and pasting code. We later found that this issue was more common than we had expected in student code, as discussed in Sec. 4.2.

2.5 Batch Code Checker

In the process of developing the extension, we had to decide on which issues we should look for in student code so that students receive the most useful information. Initially, we leveraged our own experience and developed code for immediate value checks, label checks, and so on. Later, we manually checked student code looking for common errors and borrowed insights from other TAs. However, to evaluate the usefulness of the checks that we implemented, we needed a way to run the static checker on many files at a time and generate the statistics, rather than opening thousands of files by hand.

This incentive motivated us to write an emulated client to interact with the server. The emulated client takes a directory as an argument, passes the assembly code files in the directory one-by-one to the server, collects all the feedback information, and puts the feedback into separate files in another directory. A Python script invokes the emulated client, extracts data from the feedback messages, and produces a report on how many errors, warnings, and hints for unnecessary instructions are issued for each code in the directory.

ID	Error	Warning	Unnecessary	Illegal instructions
cc	0	2	0	0
cc_complicated	0	1	3	0
code overlap	0	2	0	0
control flow	0	2	0	0
data execute	0	2	0	0
dead code	0	0	4	0
illegal	5	0	0	5
immediate	1	1	0	0
label	4	0	2	0
ORIGEND	2	2	1	0
PC offset	8	3	0	0
subroutine	0	2	0	0
uncalled subroutine	0	0	5	0
unreachable	0	0	1	0
unrolled loop	0	5	0	0

Figure 2.2: Example report generated by the batch code checker.

Figure 2.2 shows the first five columns in the report generated for the test files we used for the extension, in which we intentionally write broken code with issues to test whether the check works. The first column is the name of each file, the second to the fourth columns are the number of errors, warnings, and unnecessary instructions found in each file. The other columns correspond to specific issues. Here we only show the fifth column, which is the number of illegal instruction errors.

Using this bulk analysis scheme, we were able to evaluate the utility of our warning analyses. Code samples were available throughout students' development of each assignment, but each sample corresponds to code committed to the student's repository. Errors are thus implicitly infrequent in these samples, since students often try to assemble before committing. We expect any useful warning analysis to detect many issues in student code. Analyses that do not meet that expectation are examining problems that are rarely produced by students.

CHAPTER 3

DESCRIPTION OF ANALYSES

Static checking of code requires the ability to "understand" that code. To enable static checking on LC-3 code, we designed several rules based on frequent bugs made by students and on common requirements for elementary programming assignments. A few decisions are worth mentioning.

3.1 Immediate Values

Several LC-3 instructions have immediate value fields [1]. If an instruction in assembly code contains an immediate value outside the allowed range, the code cannot assemble. The list of immediate value ranges in LC-3 is shown in Tab. 3.1.

Table 3.1: Immediate Value Ranges for LC-3 Instructions.

Opcode	**Encoding Bits**	**Range**
ADD	5	[-16, 15]
AND	5	[-16, 15]
STR	6	[-32, 31]
LDR	6	[-32, 31]

It is easy to fix the error if the immediate value is much larger than the permitted value. For example, errors like "ADD R0, R0, x100" are easy to fix — just replace one ADD with a few ADDs, or use another register to perform the calculation.

However, there are subtle cases. The assembler is designed to be accommodating of different styles, allowing programmers to specify values as signed or unsigned.

Code such as "ADD R0, R0, #31" is not considered an error by the assembler, because the encoding of -1 and 31 is the same for a 5-bit immediate value. Numerous students write "ADD R0, R0, #16" at least once in their MP. In this case, however, it is likely to be a mistake rather than intentional. Some recognize their mistake immediately, while others spend minutes, or even hours, to find out that this instruction adds -16 to R0. Unless there is a comment saying that the programmer knows what they are doing, one can almost confidently say that the student is trying to add 16 to R0, instead of subtracting 16 from R0.

According to the reasoning above, we made the following decisions: when the opcode of an instruction is ADD or AND, raise an error when the value falls out of the range of [-16, 31] and raise a warning only if the student writes a decimal value in the range [16, 31]. Hexadecimal values do not generate warnings, as the student presumably knows which bits they have chosen to use in such cases.

3.2 PC Offsets

Load (LD), Store (ST), Load Indirect (LDI), Store Indirect (STI), and Load Effective Address (LEA) instructions are widely used for memory accesses. All five instructions use a 9-bit PC-relative offset to access the specified memory address. Similarly, Jump to Subroutine (JSR) uses an 11-bit PC-relative offset. In a long program, in which a label is far from the instruction referencing it, the PC offset may be too large. It is likely that students only realize this problem when they finish writing a great portion of the program and try to assemble the program. Unfortunately, they need to manually rearrange the code to make the LABEL closer. We raise a warning right away as they type the instruction so that they can modify the code earlier, preventing additional work to rearrange their code.

Some students write hard-coded PC offsets in their code. This style is error-prone and makes the code substantially harder to understand. Therefore, we raise a warning when students use hard-coded PC offsets in their code, despite the correctness of the code.

3.3 Unrolled Loop

```
1    LD R1, ADDRESS_1
2    BRz CHECK_1
3    STR R3,R1,#0
4  CHECK_1
5    LD R1, ADDRESS_2
6    BRz CHECK_2
7    STR R3,R1,#0
8  CHECK_2
9    LD R1, ADDRESS_3
10   BRz CHECK_3
11   STR R3,R1,#0
12 CHECK_3
13   LD R1, ADDRESS_4
14   BRz CHECK_4
15   STR R3,R1,#0
16 CHECK_4
17   LD R1, ADDRESS_5
18   BRz CHECK_FINISHED
19   STR R3,R1,#0
20 CHECK_FINISHED
```

Figure 3.1: Canonical example of a fully-unrolled loop typical of those produced by students not yet able to formulate iterative constructs.

Loop unrolling usually refers to the technique of replicating a loop body by a compiler to gain performance at the cost of code size. Here, we are not talking about this technique. Rather, novice programmers often copy and paste loop bodies because they are unable to formulate loop control. The intention of our analysis to warn students about unrolled loops is to encourage them to design a loop.

While inspecting student code, we discovered that many students have unrolled loops in their code, including numerous 5-iteration loops (over weekdays in our assignments) written as five nearly identical code segments. The length of the code segments differs from student to student, ranging from 3 lines to 28 lines. A canonical example of an unrolled loop is given in Fig. 3.1. Warnings are shown at the start of each iteration, showing the pattern clearly so that students can use this information to help them design a loop.

In one student's code, we found this comment: "I find it difficult to use a loop based on only 6 registers..." The comment indicates that the student knows that they should write a loop, but failed to do so. In order to help students design a loop, we want to provide the following information to the students: the length of the loop, the count of loop iterations, and the starting point of each iteration.

To be able to identify unrolled loops, the extension scans through the code with varying stride and tries to find repeated code segments that can potentially form a loop. The rules for an unrolled loop are as follows:

1. The instruction types must be the same for each iteration.
2. Register references must be the same.
3. Immediate values or offsets must form an equal difference sequence (including constant sequences) or a geometric sequence with ratio=2.
4. Memory accesses must form an equal difference sequence (including constant sequences).

For example, if in the first and second iterations, the immediate values are 1 and 2, respectively, in the third iteration, the immediate value of the instruction at the same position in the loop body is expected to be 3, forming the sequence [1, 2, 3] or 4, forming the sequence [1, 2, 4]. The insight behind this approach is that one can acquire the sequence from a loop control variable easily by doing addition or shifts. Violation of these rules may add difficulty to converting the code a student has written into a loop. Therefore, we chose not to raise a warning in such cases.

3.4 Control Flow Analysis

Control flow analysis is an important part of static analysis, which includes the identification of conditional structures, loop structures, and subroutines. As an assembly language, LC-3 does not have keywords such as if, else, for, or while. To be able to identify different control structures is also essential for some other analyses.

We implemented a forward-backward control flow analysis, which can be decomposed into the following steps:

Step 1: Build a Control Flow Graph The first step involves linking each instruction to the instructions that might follow it. Most instructions have one or two possible next instructions, while some instruction types, such as RET and JMP, may have many. [1]

[1] We ignore RTI here, as the programming assignments in our class never make use of it.

We connect each instruction with the instruction at the next memory location. We call this linkage **Next Instruction Linkage**. There are exceptions, namely JMP (including RET), BRnzp, and HALT, which always change the control flow, so we do not create this link for those instructions.

Then we handle control instructions according to the way that they affect control flow.

For conditional branch instructions, there are two possible next instructions. Another linkage is set up in this case to connect the branch instruction with the **Branch Target**. [2]

For jump to subroutine (JSR) instructions, we assume that a subroutine always returns properly, so, after execution of the subroutine, control returns to the next instruction in memory. For this reason, we keep the **Next Instruction Linkage** for JSR instructions. We also add another linkage, **Subroutine Start**, to connect each JSR instruction with the called subroutine. At the same time, a flag indicating the start of a subroutine is set on the first instruction in the called subroutine. The flag is used in the next step.

For jump (JMP) [3] instructions, given the difficulty of analyzing the exact values in all the registers without input information, we support only those jumps that immediately follow a load (LD) instruction. Those two instructions effectively form an unconditional branch with arbitrary range.

In that case, we update the **Next Instruction Linkage** to be the destination address stored in the source of the LD instruction. An assumption is made here: that the destination address is constant throughout the execution of the program. The assumption may not always be true, but is our best guess without resorting to dynamic analysis.

For system call (TRAP) instructions, the predefined trap pseudo-operations are GETC, OUT, PUTS, IN, PUTSP and HALT. Except for HALT, we can view each TRAP instruction as one instruction performing some tasks, so nothing needs to be done.

[2] Although one can encode a NOP (no operation) using a branch with no condition bits in LC-3, one cannot do so as a branch (BR) instruction in LC-3 assembly, so we do not need to consider such cases.

[3] Although RET is equivalent to JMP R7 in encoding, we do not consider RET here. Also, "JMP R7" is not considered as a RET during the analysis.

We allow students to write TRAP instructions in both forms. For example, "TRAP x25" and "HALT" are equivalent. We currently do not support the analysis of user-defined TRAPs. Undefined TRAP vectors raise errors.

Step 2: Check Reachability In this step, we try to step through the code to examine the reachability of all instructions using a depth-first search (DFS). A DFS is performed for the main routine and for each subroutine. The information obtained from JSR instructions in Step 1 is used in Step 2 to identify subroutine entry points.

The DFS is performed in the following way. We first initialize the stack to contain only the instruction at the entry point. This step is done for the main routine and for each subroutine exactly once. Then, we loop until there are no instructions on the stack. The loop body involves popping the instruction on top of the stack, then pushing both the **Next Instruction** and the **Branch Target** of the popped instruction if they have not yet been explored, then marking each as explored. **Subroutine Start** links are added to the list of subroutines to be explored, but are not explored during the DFS of the caller.

We also record a subroutine number for each explored instruction. The number that we use is the memory address of the entry instruction, which acts as a unique ID for each subroutine. The subroutine number for instructions in the main routine is the program start point indicated by the .ORIG directive. If an instruction has been explored before, we examine whether the subroutine numbers match. If not, it means that there is code overlap between the main routine and a subroutine, or between different subroutines.

After Step 2 is finished, the explored flag and subroutine numbers are set for all instructions. Any instructions that are not explored are unreachable because there is no path to reach the instruction from the starting point of the main routine nor from any executed subroutine.

Step 3: Build Blocks In this step, basic blocks are built out of connected instructions.

Basic Blocks correspond to sequential structures with a single entry point in the control flow of the code, that is, a series of consecutive instructions ending with a control instruction. The basic blocks are built recursively once for the main routine and once for each subroutine.

Each block keeps two internal fields as interconnecting linkage, called **Next Block** and **Branch Block**. **Next Block** corresponds to the block executed after the current block if the branch at the end of the block is not taken, and **Branch Block** corresponds to the block executed after the current block if the branch is taken. The last blocks of the main routine and subroutines have neither **Next Blocks** nor **Branch Blocks** (these fields are set to null pointers).

Step 4: Analyze Blocks In the final step, we analyze the basic blocks internally and externally to detect issues such as dead code and redundant condition codes. This step is composed of two analysis flows applied repeatedly. A forward flow performs condition code checking, and a backward flow identifies register usage to detect dead code.

The backward flow checks for dead code by keeping track of the read/write status of all registers. For example, if a register is written to twice without reading the value in it, the first write instruction effectively does nothing. Removing the first instruction has no effect on the output of the code. It is a common issue to have dead code, but for assembly language, having dead code usually means that the student does not have a clear understanding of what they are doing, which may indicate a bug.

For TRAP instructions, we handle different TRAP vectors according to their predefined behaviors. Table 3.2 shows the register use semantics for all predefined TRAP vectors.

Table 3.2: Register Operations Involved in TRAP Instructions.

TRAP Vector	**Register Operation**
GETC	write R0, write CC
OUT	read R0, write CC
PUTS	read R0, write CC
IN	read R0, write CC
PUTSP	read R0, write CC
HALT	none (halts machine)

As an example of dead code, consider Fig. 3.2, in which the programmer intended to put the value R3 - R2 into R1, but mistakenly wrote R3 in place of R1. In this case, the value (R3 - R2) calculated by the first two instructions is discarded when the third instruction executes.

We indicate that the first two instructions are dead code by marking them as unnecessary. By examining this code carefully, the programmer should be able to identify the bug.

```
1    NOT R1, R2
2    ADD R1, R1, #1
3    ADD R1, R3, R3   ; Should be ADD R1, R1, R3
```

Figure 3.2: A simple example of dead code.

The forward pass concentrates on checking Condition Codes (CC). The LC-3 Condition Code registers are three 1-bit registers that record the condition codes needed for conditional branch instructions. These three registers represent mutually-exclusive cases: Negative (N), Zero (Z), and Positive (P).

The CC right after a taken branch are always within the subset of the CC specified in the branch instruction. On the other hand, if a branch is not taken, the CC are in the complementary set of the branch's conditions.

Our analysis identifies all possible CC at the start of each basic block by taking the union of all possible incoming CC. If there are instructions that modify CC, we reset the possible set to be NZP. Then we compare all possibilities of CC with the CC specified by the branch instruction at the end of the basic block, if any, to determine whether the branch is either: 1) always taken but not written as BR or BRnzp, or 2) never taken. If either case happens, we remove the **Next Instruction Linkage** or **Branch Target**, respectively, in the corresponding branch instructions. By doing so, we can identify the impossible control paths in the code, increasing the power of checks for unreachable instructions.

```
1   .ORIG x3000
2
3   ; TEST: Complicated condition code
4   START
5     BRn LABEL
6   MIDDLE
7     BRn IMPOSSIBLE
8   LABEL
9     BRnzp END
10  IMPOSSIBLE
11    BRnp MIDDLE
12    HALT
13
14  END
15    HALT
16
17  .END
18
```

Figure 3.3: Example of a complicated condition code issue.

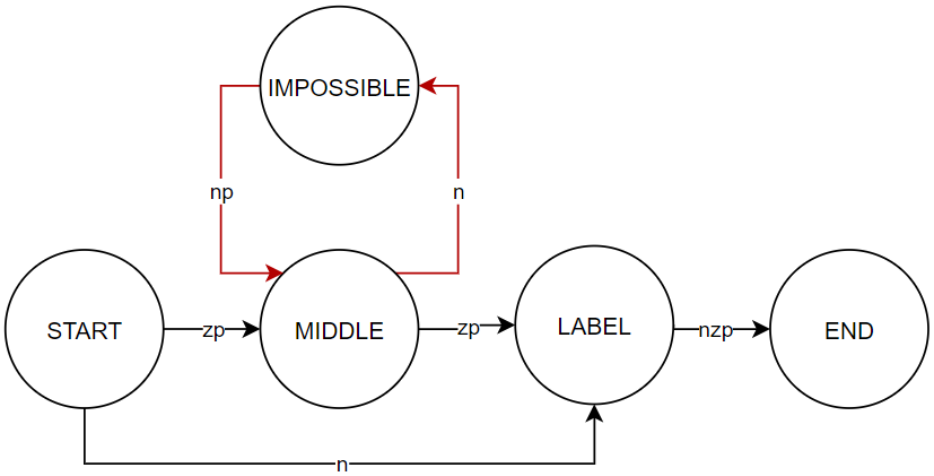

Figure 3.4: Control flow graph for the code example in Fig. 3.3.

In the example code shown in Fig. 3.3, the CFG for the code snippet is built as Fig. 3.4. Each basic block is a node (a circle) in the graph. The n, z, and p labels on the paths (represented as arcs between the basic blocks) indicate under which CC the code takes the path. In the CFG, the two red paths are impossible. The reason is given below.

The only way to get to IMPOSSIBLE is to go from MIDDLE when CC are n (indicated by the red line on the right). However, the CC can never be n in MIDDLE; in order to reach MIDDLE, the CC must be z or p. With the CFG trimming done by the condition code analysis, the extension is capable of detecting such issues caused by possibly incorrect CC of branch instructions.

The other benefit that we obtain from CFG trimming is that redundant CC coupled with unrolled loops generate many impossible paths, which severely harm the speed of the KLEE-based feedback tool that we designed [4], to the point that the tool never finishes running on the code. By warning students about the issue and the potential risk, we encourage students to fix the issues before they submit their code, thus accelerating the feedback system.

These four steps are repeated a few times until convergence, meaning that there is no change in the state of any basic blocks. We also set the upper limit on iterations to be 5 to prevent dragging down performance on long programs.

3.5 Subroutine Analysis

LC-3 assignments often require students to write subroutines. The subroutine analysis is designed to provide feedback on subroutine interfaces and structure.

A subroutine has a set of callee-saved registers (values preserved by the subroutine) and caller-saved registers (values possibly modified by the subroutine). Except for R7, which is always caller-saved, we analyze the usage of all registers to identify callee-saved registers.

If a register is saved to memory (by an ST instruction) in the first basic block of the subroutine, and restored from the same memory location (by an LD instruction) in any exit block (a block ending in a RET instruction) of the subroutine, the register is callee-saved. There may be multiple exit blocks in a subroutine. In this case, we require a register to be restored on every exit block. However, we generally do not recommend that students organize their code in this way. Although a programmer can also preserve the value in a register by only wrapping around the part of code that modifies the register, we do not consider this case.

If a register is never modified in the subroutine, it is also considered callee-saved.

We also check whether the saving and restoring of registers appear as matched pairs. If a register is saved multiple times in the first basic block, or different registers are saved to the same memory location, it is likely to be a typo. Therefore, we generate a warning. Similarly, if a register is restored multiple times in an exit block, or different registers are restored from the same memory location, we also generate a warning. Also, if a register is saved but not restored, or not saved but restored, or the memory locations of saving and restoring are different, we generate a warning to tell the student that there is a mismatch, which may be a bug.

Finally, we show the callee-saved registers when a user hovers the mouse cursor over the subroutine label, so that they can verify that the set of callee-saved registers is correct.

CHAPTER 4

RESULTS

After developing the batch code checker, we ran it against all student code from Fall 2020 (3548 samples) and Fall 2018 (237 samples, final commits only) to collect the statistics of all feedback messages generated by the VSCode extension. We found a few interesting results in the data.

4.1 Grade Dependence on Warnings

First, we are able to illustrate that the ability to convey meaningful feedback messages to students while they write their code increases their expected functionality grade in assignments. Using the MP1 program as an example, for which 770 code samples assemble, we used our grading script to calculate the functionality grade (out of 65) for each sample, then computed an average functionality grade among samples for which our extension reports the same number of warnings. The results appear in Tab. 4.1.

None of our extension's feedback is specific to any particular assignment. Nevertheless, grades for samples that are warning-free average over 10 points higher than samples containing warnings, indicating that feedback during editing can be helpful in guiding students to develop correct solutions.

Table 4.1: Functionality Grades vs. Number of Warnings.

Number of Warnings	**Count**	**Avg. Functionality Grade**
0	492	46.9
1	169	36.0
2	42	40.9
3+	67	35.9

4.2 Student Behavior Dependence on Problem Specification

We found that hand-unrolled loops are common in student code, and that their frequency depends strongly on the complexity of the particular loop that students are asked to write. To illustrate this idea, we compared the final versions of the MP3 assignment of students in the Fall 2018 semester with those of the Fall 2020 students. The assignment was changed only in minor ways to reduce the likelihood of sharing code between semesters. In particular, the days of the week were printed as three-letter abbreviations in 2018, but as full names in 2020. Also, the encoding of days in the bit vector for each event was reversed: in 2018, Monday was represented as 1, Tuesday as 2, and so forth. In 2020, Monday was 16, Tuesday was 8, and so forth.

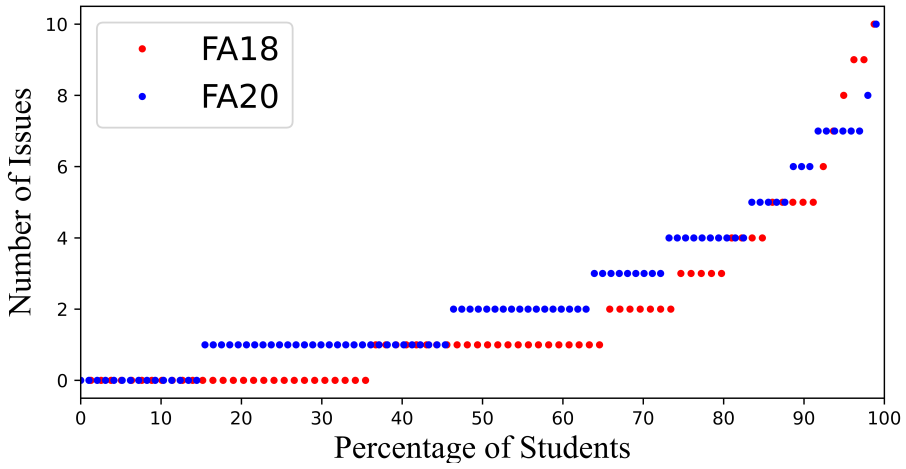

Figure 4.1: Number of unrolled loops found in student code in MP3.

The results are shown in Fig. 4.1 and summarized in Tab. 4.2: failure to write loops is more common among the 2020 students, but is generally common in both classes. In terms of the assignments, the slight changes produced visible differences in the results by changing the complexity required to conceptualize loops. Specifically, the variable-length weekday names complicate the process of finding the starting address of each string, and the reversal of bit vector ordering makes using these data more challenging because the LC-3 ISA makes left shift easy, but right shift difficult.

Table 4.2: Percentage of Students vs. Number of Unrolled Loops.

Number of Unrolled Loops	**Fall 2018**	**Fall 2020**
0	36%	14%
1	29%	31%
2+	35%	55%

Fortunately, we were able to add an analysis to our VSCode extension that identifies hand-unrolled loops and raises warnings to encourage students to think harder or to seek help for implementing a loop before submitting their code, as described in Sec. 3.3.

4.3 Usage of the Extension

Do students made aware of potential issues in their programs try to address those issues before turning in the code? One of our hopes in developing the extension, of course, is that the answer is yes. By comparing the issues reported for code samples from students in the Fall 2018 semester with those reported for student samples from 2020, we were able to validate this hypothesis.

The same instructor (Prof. Lumetta) taught both the Fall 2018 and Fall 2020 sections of the course, and the assignments were nearly identical. One might reasonably still expect a fair bit of variation due to other factors, such as differences among the students themselves and differences in the other course staff[1].

However, we are able to use analyses developed after the Fall 2020 semester as experimental controls to illustrate that the outcomes are largely identical across the classes when our extension either is unavailable or does not report a particular type of issue.

[1]There was no overlap, although students from the Fall 2018 section did serve as the TAs in 2020.

Table 4.3: Comparison of Number of Issues across Two Semesters.

		Fall 2018			Fall 2020		
	Issue	**0**	**1**	**2+**	**0**	**1**	**2+**
Available	Multiple Label	62%	12%	16%	88%	1%	11%
in Fall 2020	Code Overlap	68%	17%	15%	94%	2%	4%
Not available	Condition Code	57%	18%	25%	59%	20%	21%
in Fall 2020	Dead Code	17%	15%	68%	15%	14%	71%

Since the development of the extension and the offering of the course took place at the same time, and we did not want to release an incomplete or faulty version of the extension, we decided to distribute the extension only using offline packages. Therefore, students may not have updated the extension to the latest version during the semester, especially after they had finished the programming assignments.

We found that the Fall 2020 students as a whole have fewer warnings compared to Fall 2018 class in certain categories, for example, multiple labels at the same memory location and code overlap, shown in the first two rows in Tab. 4.3. However, in some other categories, like condition code issues and dead code, there is no major difference, as shown in the last two rows. Detailed graphs for each distribution can be found in Appendix B.

We only implemented the condition code check and dead code check in a later version of the extension, by which time most students had finished the third assignment. They probably did not use the version with those checks. The differences in those two types of data show that the students were actively using the extension and that students made aware of issues in their code have a higher tendency to fix the issues before handing in the assignments.

CHAPTER 5

CONCLUSION

Our VSCode extension for LC-3 programming is effective in detecting errors, stylistic issues, and potential bugs in student code. The extension assists students in introductory programming courses in developing a good coding style as well as warning them of common bugs found in samples of previous students' code. The extension can also be used by instructors in batch mode to reduce the time needed to grade coding style.

More data is needed to confirm that the extension is helpful for not only a specific class, but generally for any class using LC-3 assembly language. Also, as the LC-3 ISA was slightly modified in the recent new edition of Patt and Patel's textbook [11], the extension should be updated accordingly.

APPENDIX A

EXAMPLE CODE AND FEEDBACK MESSAGES

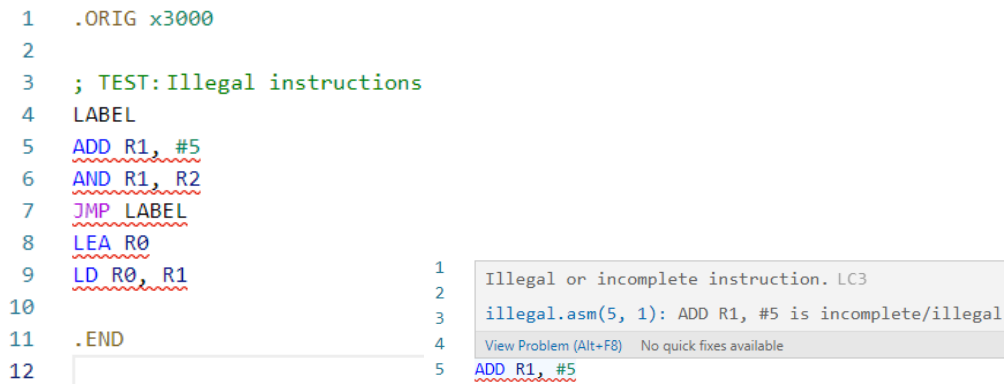

Figure A.1: Example of illegal instruction check.

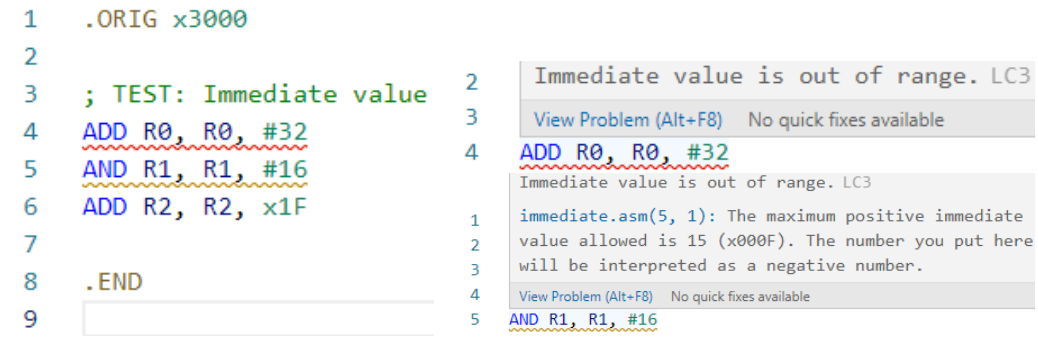

Figure A.2: Example of immediate value check.

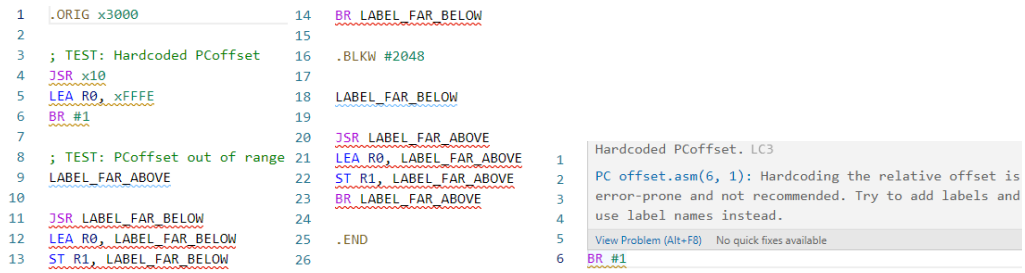

Figure A.3: Example of PC offset check.

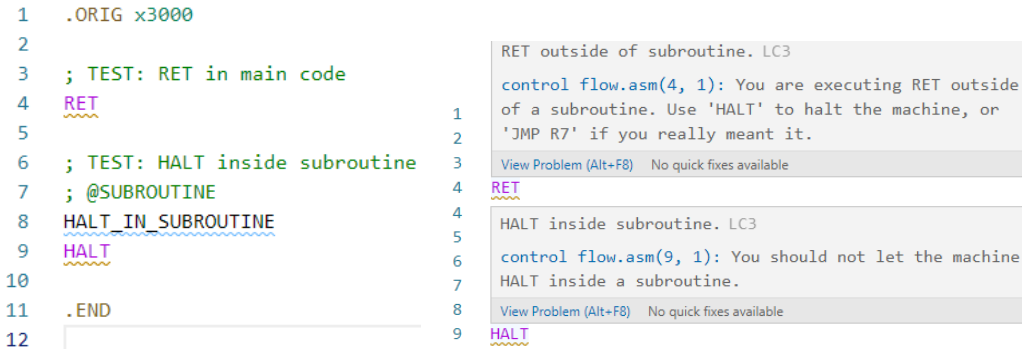

Figure A.4: Example of control flow check.

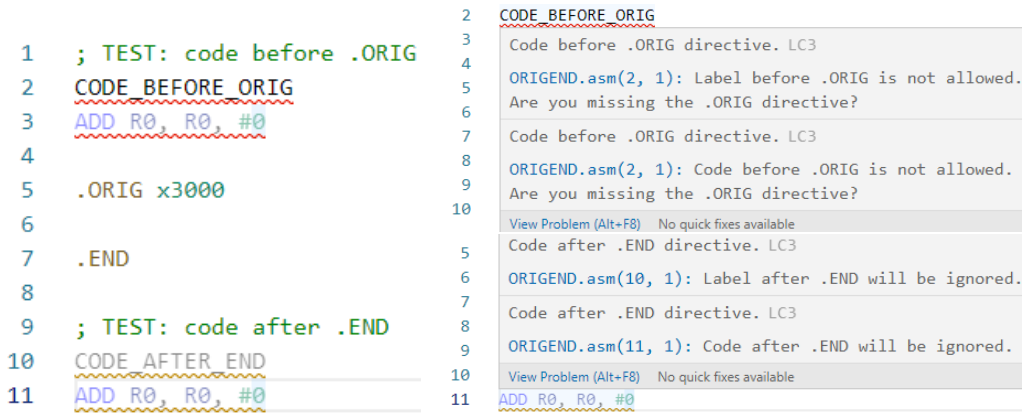

Figure A.5: Example of start address and end of file check.

Figure A.6: Example of data execution check.

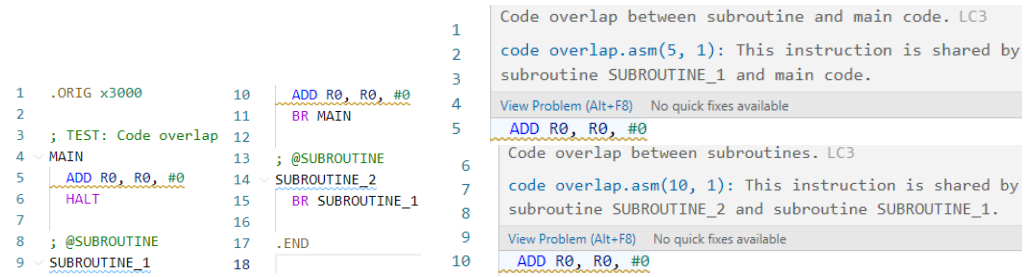

Figure A.7: Example of code overlap check.

Figure A.8: Example of label check.

Figure A.9: Example of condition code check.

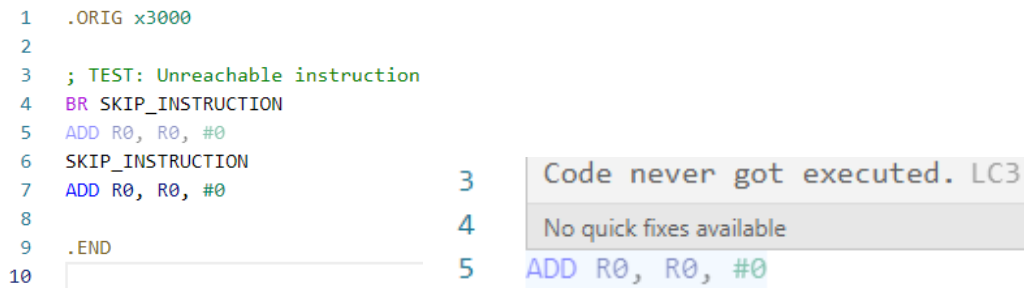

Figure A.10: Example of unreachable instruction check.

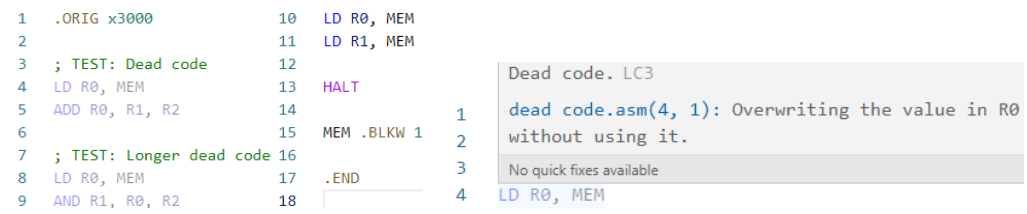

Figure A.11: Example of dead code check.

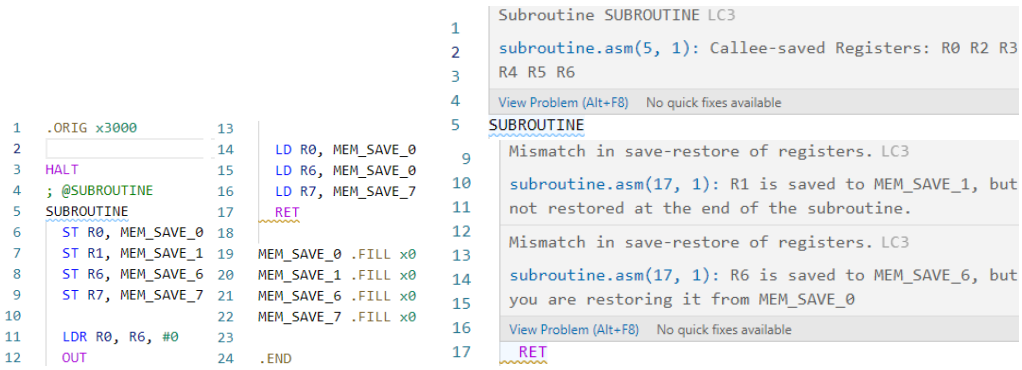

Figure A.12: Example of subroutine analysis.

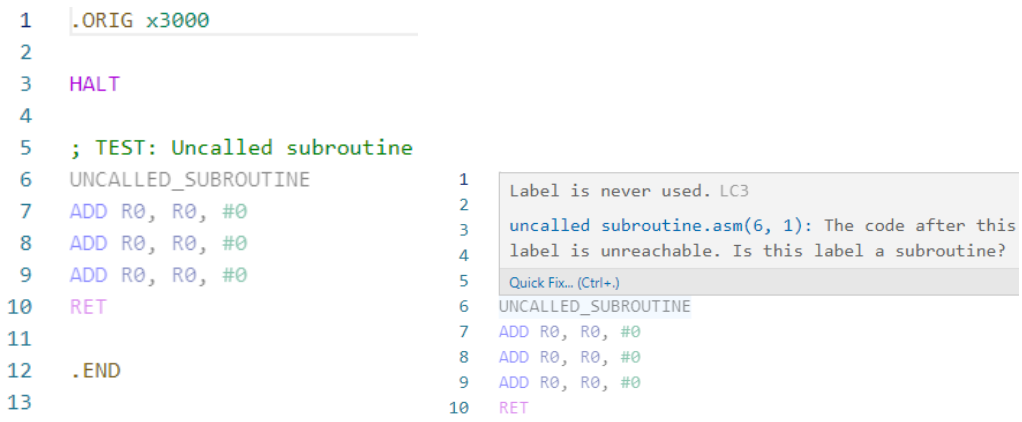

Figure A.13: Example of uncalled subroutine check.

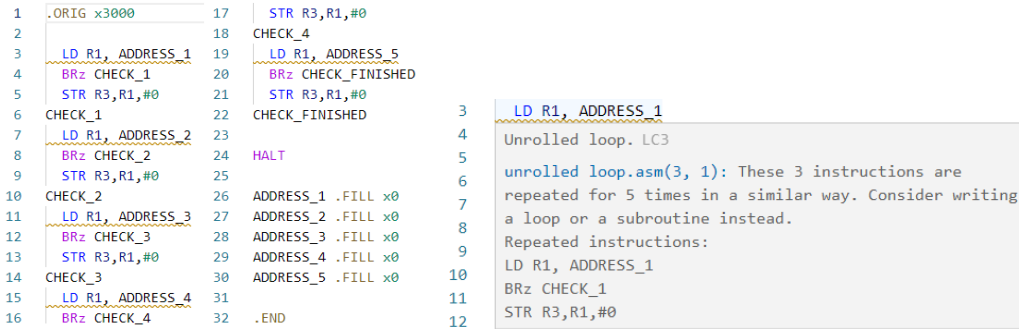

Figure A.14: Example of unrolled loop check.

APPENDIX B

PLOTS OF ISSUES IDENTIFIED IN MP3

Considering different class sizes in Fall 2018 and Fall 2020, we normalize the student number to 100 for both semesters.

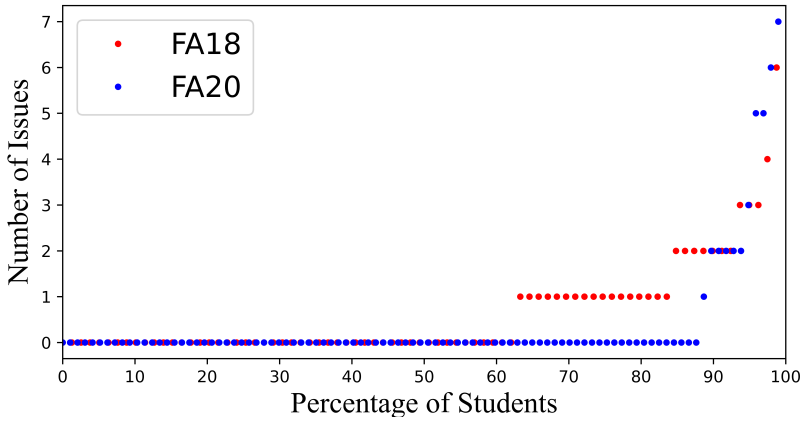

Figure B.1: Number of multiple label issues identified for the final commits of MP3 in the Fall 2018 and Fall 2020 semesters (Available only in Fall 2020).

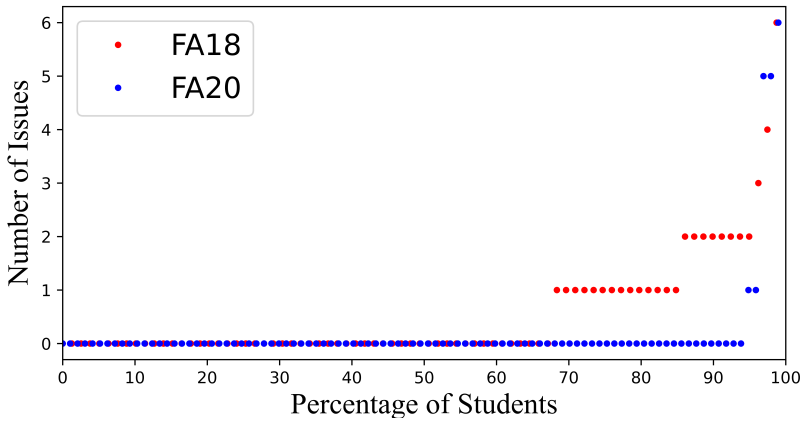

Figure B.2: Number of code overlap issues identified for the final commits of MP3 in the Fall 2018 and Fall 2020 semesters (Available only in Fall 2020).

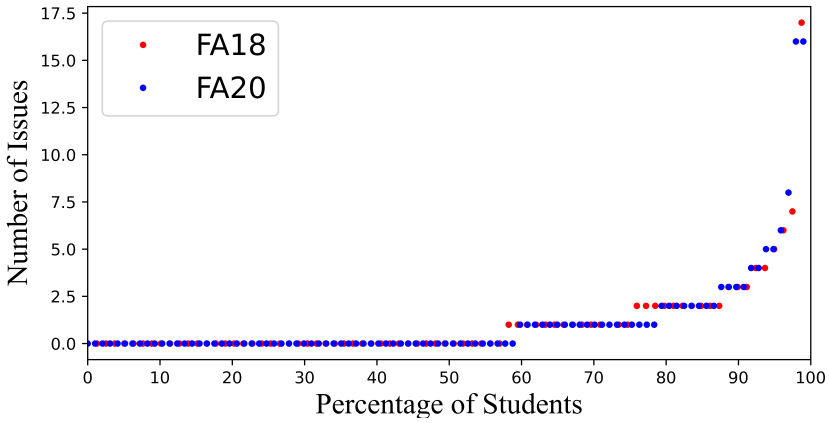

Figure B.3: Number of condition code issues identified for the final commits of MP3 in the Fall 2018 and Fall 2020 semesters (Not Available in Both Semesters).

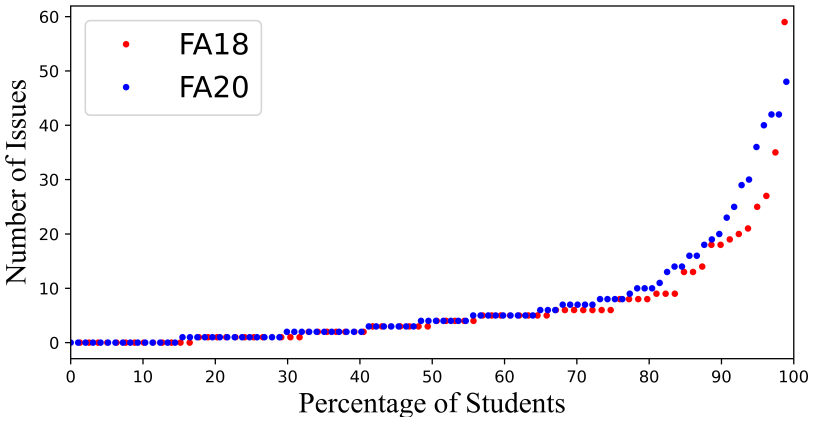

Figure B.4: Number of dead code issues identified for the final commits of MP3 in the Fall 2018 and Fall 2020 semesters (Not Available in Both Semesters).

REFERENCES

[1] Y. Patt and S. Patel, *Introduction to Computing Systems: From Bits and Gates to C and Beyond*, 2nd ed. McGraw-Hill Higher Education, 2004.

[2] "Visual Studio Code: Code Editing. Redefined." https://code.visualstudio.com/.

[3] "VSCode Extension for LC-3." https://marketplace.visualstudio.com/items?itemName=qili.vscode-lc3.

[4] Z. Liu, "Using concolic execution to provide automatic feedback on LC-3 programs," B.S. thesis, University of Illinois at Urbana-Champaign, 2021.

[5] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[6] "Git." https://git-scm.com/.

[7] W. Luo, "In-browser LC-3 toolchain and queue management for symbolic testing," B.S. thesis, University of Illinois at Urbana-Champaign, 2021.

[8] T. Liu, "Improved feedback and debugging support for student assembly programming," B.S. thesis, University of Illinois at Urbana-Champaign, 2021.

[9] "Visual Studio Code Extension API." https://code.visualstudio.com/api.

[10] "Language Server Protocol." https://microsoft.github.io/language-server-protocol/.

[11] Y. Patt and S. Patel, *Introduction to Computing Systems: From Bits and Gates to C and Beyond*, 3rd ed. McGraw-Hill Higher Education, 2020.